Functions, Methods, and Lambdas

PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

Overview

- function as a parameter
- method as a parameter
- lambda
  - definition
  - capture list
  - as a parameter
- C++ libraries
  - algorithm library
  - iterator library
- lazy library

Overview

- function as a parameter
- method as a parameter
- lambda
  - definition
  - capture list
  - as a parameter
- C++ libraries
  - algorithm library
  - iterator library
- lazy library
  - . . . oh wait, that was the homework

Motivation

Why we do want it?

Motivation

Why we do want it?
Cause it is cool!

Motivation

Why we do want it?
Cause it is cool!

Do we need it?

Motivation

Why we do want it?
Cause it is cool!

Do we need it?
No, but C++ without lambdas as a concept would be like Java without classes or Haskell without functions.

Function as a parameter

```
int foo( int a, int b ) {
    return a * 3 + b;
}
```

Function as a parameter

```
int foo( int a, int b ) {
    return a * 3 + b;
}
```

How to create a pointer to function?

Function as a parameter

```
int foo( int a, int b ) {
    return a * 3 + b;
}
```

How to create a pointer to function?

```
int main() {
    auto f = foo;
    std::cout << f( 3, 8 ) << std::endl;
}
```

What is the type of `f`?

Function as a parameter

```
int foo( int a, int b ) {
    return a * 3 + b;
}
```

The type of the `foo` function is `int( int, int )`

Function as a parameter

```
int foo( int a, int b ) {
    return a * 3 + b;
}
```

The type of the `foo` function is `int( int, int )`

```
using FooType = int( int, int );
// FooType is NOT a pointer
FooType *ptrToFoo = foo;
int(*ptrToFoo2)(int, int) = foo;
```

Function as a parameter

There was a question:

> *Why do function pointer definitions work with any number of ampersands '&' or asterisks '*'?*

Function as a parameter

There was a question:

> *Why do function pointer definitions work with any number of ampersands '&' or asterisks '*'?*

```
void foo() { cout << "Foo to you too!\n"; }

int main() {
    void (*p1)() = foo;  void (*p2)() = *foo;
    void (*p3)() = &foo; void (*p4)() = *&foo;
    void (*p5)() = &*foo;void (*p6)() = **foo;
    void (*p7)() = **********************foo;
    (*p1)(); (*p2)(); (*p3)();
    (*p4)(); (*p5)(); (*p6)();
    (*p7)();
}
```

Function as a parameter

Explanation

- expression `foo` is implicitly convertible to a pointer to the function
- expression `*foo` results to `foo`
- expression `&foo` takes an address of the function (i.e. a pointer to the function)
- expressions can be combined together

Method as a parameter

```
struct X {
    int foo( int a, int b ) {
        return a * 3 + b;
    }
};
```

Method as a parameter

```
struct X {
    int foo( int a, int b ) {
        return a * 3 + b;
    }
};
```

How to create a pointer to a member function?

Method as a parameter

```
struct X {
    int foo( int a, int b ) {
        return a * 3 + b;
    }
};
```

How to create a pointer to a member function?

```
int main() {
    X x;
    auto f = &X::foo;
}
```

- What is the type of `f`?
- How can we call `f`?
- Is the ampersand necessary?

Method as a parameter

- What is the type of `f`?
  - `int (X::*)( int, int )`

Method as a parameter

- What is the type of `f`?
  - `int (X::*)( int, int )`

- How can we call `f`?
  - `(x.*f)( 3, 8 )`
  - `(ptrToX->*f)( 3, 8 )`

Method as a parameter

- What is the type of `f`?
  - `int (X::*)( int, int )`
- How can we call `f`?
  - `(x.*f)( 3, 8 )`
  - `(ptrToX->*f)( 3, 8 )`
- Is the ampersand necessary?
  - Yes. Rules for taking address of member function are different to the old C rules for plain functions.
  - No. Just for Visual Studio.

Lambda – definition

```
[capture list] ( parameters ) -> return_type { body }
```

- capture list
  - which variables from outside should be visible in the lambda

Lambda – definition

```
[capture list] ( parameters ) -> return_type { body }
```

- capture list
  - which variables from outside should be visible in the lambda
- parameters
  - the same list as in the functions
  - ... or `auto` as a type
  - can be ommited if the list is empty

Lambda – definition

```
[capture list] ( parameters ) -> return_type { body }
```

- capture list
  - which variables from outside should be visible in the lambda
- parameters
  - the same list as in the functions
  - . . . or `auto` as a type
  - can be ommited if the list is empty
- return type
  - it is what it represents. . .
  - can be ommited if the type is *obvious*

Lambda – definition

```
[capture list] ( parameters ) -> return_type { body }
```

- capture list
  - which variables from outside should be visible in the lambda
- parameters
  - the same list as in the functions
  - ... or `auto` as a type
  - can be ommited if the list is empty
- return type
  - it is what it represents...
  - can be ommited if the type is *obvious*
- body
  - the code

Lambda – capture list

- list of variables
  - with ampersand – references
  - without ampersand – copies
    - `const`
- `this`
- & – capture all as a reference
- = – capture all by copy
  - `const`
- initializer
  - introducing new variable

Lambda – capture list

Examples

```
[=,this,&events] {
    ++events;
    return this->foo( a ) + b;
}
```

Lambda – capture list

Examples

```
[=,this,&events] {
    ++events;
    return this->foo( a ) + b;
}
```

```
int x = 4;
int y = [&r = x, x = x + 1] {
    ++r;
    return x + 1;
}();
```

Lambda as a parameter

What is the type of lambda?

Lambda as a parameter

What is the type of lambda?
Who knows. . . *(The compiler knows.)*

Lambda as a parameter

What is the type of lambda?
Who knows. . . *(The compiler knows.)*

How to store lambdas?

Lambda as a parameter

What is the type of lambda?
Who knows. . . *(The compiler knows.)*

How to store lambdas?

- `auto`
- template parameter
- `std::function< signature >`
  - runtime overhead
- pointer to function
  - requires empty capture list

Lambda as a parameter

Example

```
void e1( void(*f)( int ), int p ) { f( p ); }

template< typename F >
void e2( F f, int p ) { f( p ); }

void e3( std::fucntion< void(int) > f, int p ) {
    f( p );
}

auto foo = []( int i ) { std::cout << i; };
e1( foo, 1 );
e2( foo, 1 );
e3( foo, 1 );
```

How to use lambdas

- Do not overuse them.
- Lambdas should be short. Four lines at maximum.
- If you need to name the lambda, use a function instead.
- If your capture list is long, choose a different approach.
- If your lambda is long, use a function or a method instead.
- Prefer references to copies in the capture list.
  - The generated class will be smaller.
  - References could be dangerous. Be careful.
- If the number of lambdas is higher than number of methods, you should consider refactoring your code.

C++ libraries

- algorithm library
  - copy, transform, generate
  - remove, reverse, fill
  - equals, find, count
- iterator library
  - `back_inserter`
  - `istream_iterator`
  - `ostream_iterator`

Exercise 1

Implement template function `forEach` so that:

- takes two input iterators
  - first and last
- takes a function callback as a third parameter
  - by pointer to function
  - by template parameter
  - by `std::function`
- use a simple lambda
  - increment parameter, multiply parameter, . . .

```
template< typename It, /*...*/ >
void forEach( It first, It last, /*...*/ f ) { /*...*/ }
```

Compare the speed on large container.
Use SequenceGenerator from study materials.

Exercise 2

Refactor *03_lines.cpp* so that:

- reading of lines is placed in a new function
- printing is realized by a lambda function

```
template< typename F >
void readLines( const char *file, F f ) {
    // ...
}
```

Exercise 2

Refactor *03_lines.cpp* so that:

- reading of lines is placed in a new function
- printing is realized by a lambda function

```
template< typename F >
void readLines( const char *file, F f ) {
    // ...
}
```

- change behaviour to print only even lines

Exercise 3

Refactor *03_algorithm.cpp* so that:

- no explicitly written cycle is present
- function `almostSame` works for all containers
  - not just those with the random access
- use constructs from
  - algorithm library
  - iterator library